CSD2183: Data Structures

Disjoint Sets and Minimum Spanning Trees

Bingjie Xu

Contact: bingjie.xu@singaporetech.edu.sg

2 March 2026

Information

- The midterm exam grades will be released today on Gradescope.
- I have organized additional learning resources on NotebookLM for deep dive. The link is available on xSITe content. Please explore them, ask questions, and feel free to share your feedback and experience with me.

Overview

	Basics (Week 1 - 6)	**Advanced (Week 8 - 13)**
1	Foundations	Graph Foundations and Traversal
2	Running Times	**Disjoint Sets and Minimum Spanning Trees**
3	Sorting	Shortest Paths
4	List and Hash Tables	Dynamic Programming
5	Trees	Greedy Algorithms
6	Consultation	Consultation

Cormen, Thomas H., et al. Introduction to algorithms 4th edition. MIT press, 2022.

Learning Objectives

By end of this lecture, you will be able to:

- Apply union-find operations to disjoint-set forests.
- Explain what a minimum spanning tree is.
- Understand and apply Kruskal’s and Prim’s algorithms to a given graph.
- Analyze the growth of the running time of Kruskal’s and Prim’s algorithms as a function of the number of vertices and the number of edges.

Disjoint Sets

Core structure used in MST (e.g., Kruskal's algorithm).

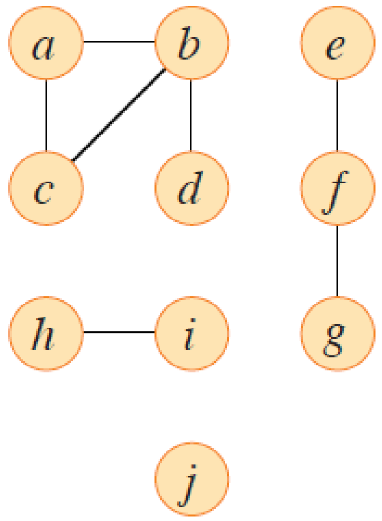
Introduction to Disjoint Sets

- Consider a dynamic graph where the edges are added one by one, and we want to know the connected components at all stages of the growth process.

- We could run DFS repeatedly after each new edge is added to the graph. However, this strategy is inefficient. Let us take a look at an example to gain intuition.

Introduction to Disjoint Sets

- Consider the vertices $a, b, \dots, j$. Without any edges, each vertex is in its own connected component (in undirected graph).
- Insert the edges (b, d), (e, f), (a, c), (h, i), (a, b), (f, g) and (b, c) one by one.
- The connected components change as the edges are inserted into the graph.



Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b, d)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e, f)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e, f\}$		$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(a, c)	$\{a, c\}$	$\{b, d\}$			$\{e, f\}$		$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(h, i)	$\{a, c\}$	$\{b, d\}$			$\{e, f\}$		$\{g\}$	$\{h, i\}$		$\{j\}$
(a, b)	$\boxed{\{a, b, c, d\}}$				$\{e, f\}$		$\{g\}$	$\{h, i\}$		$\{j\}$
(f, g)	$\{a, b, c, d\}$				$\boxed{\{e, f, g\}}$			$\{h, i\}$		$\{j\}$
(b, c)	$\{a, b, c, d\}$				$\{e, f, g\}$			$\{h, i\}$		$\{j\}$

Disjoint-Set Data Structures

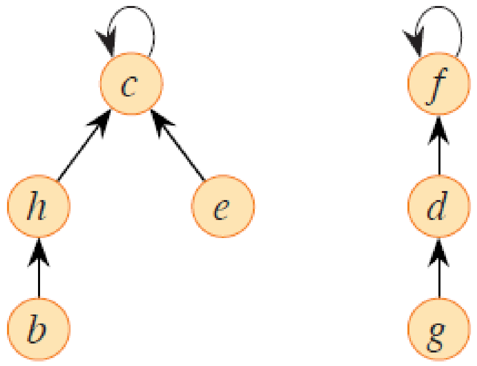
- A disjoint-set data structure maintains a collection $S = \{S_1, S_2, \dots, S_k\}$ of disjoint dynamic sets.
  e.g., $\{\{a,b,c,d\},\{e,f,g\},\{h,i\},\{j\}\}$.
- Also known as "union find".
- Each set is identified by a representative, which is some member of the set. Doesn't matter which member is the representative, as long as if we ask for the representative twice without modifying the set, we get the same answer both times.

Disjoint-Set Operations

- We represent members of the sets by objects. Letting x denote an object, we wish to support the following operations applied to objects :
  - MAKE-SET(x): make a new set $S_i = \{x\}$ and add S_i to S.
  - UNION(x, y): $if\ x \in S_x, y \in S_y$, then $S = (S - S_x - S_y) \cup \{S_x \cup S_y\}$.
    - merge two separate sets while maintain disjoint-set property.
  - FIND-SET(x): return representative of set containing x.
- We assume that all sets are disjoint before and after these operations.

Disjoint-Set Forests

- A type of disjoint-set data structure for efficient implementation of disjoint-set operations.
- Forest of trees:
  - One tree per set. Root is the representative and is its own parent.
  - Each node points only to its parent.



Disjoint-Set Forests

Implementation

Each node has two attributes: p (parent) and $rank$ (upper bound on height of node).

- **MAKE-SET**: make a single-node tree.

```
MAKE-SET($x$)
    $x.p = x$                // Make x the root by assigning itself as parent.
    $x.rank = 0$             // x is the leaf, so assign height 0.
```

Disjoint-Set Operations

Application: Dynamic connected components

For a graph $G = (V, E)$, vertices u, v are in the same connected component if and only if there's a path between them.

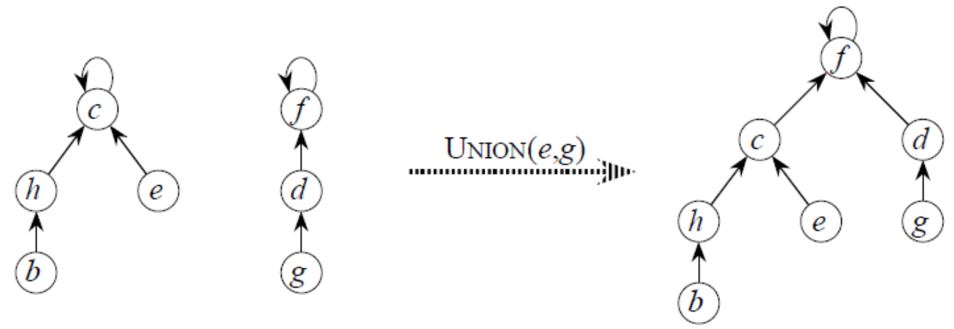
```
CONNECTED-COMPONENTS($G$)
  for each vertex $v \in G.V$                // Initializes each vertex as a separate set.
      MAKE-SET($v$)
  for each edge $(u, v) \in G.E$
      if FIND-SET($u$) $\neq$ FIND-SET($v$)   // Compare representative (root)
          UNION($u, v$)                          of the sets containing $u, v$.
```

Disjoint-Set Forests

- **UNION**: make one root a child of the other, **union by rank**.
- **Rank** is a heuristic measure of tree height.
- This keeps the tree shallow and makes operations faster.

```
UNION(x, y)
  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
  if $x.rank > y.rank$
      $y.p = x$
  else $x.p = y$
      // If equal ranks, choose y as parent and increment its rank.
      if $x.rank == y.rank$
          $y.rank = y.rank + 1$
```

// LINK (x,y) makes the root of the smaller tree (fewer nodes) a child of the root of the larger tree.



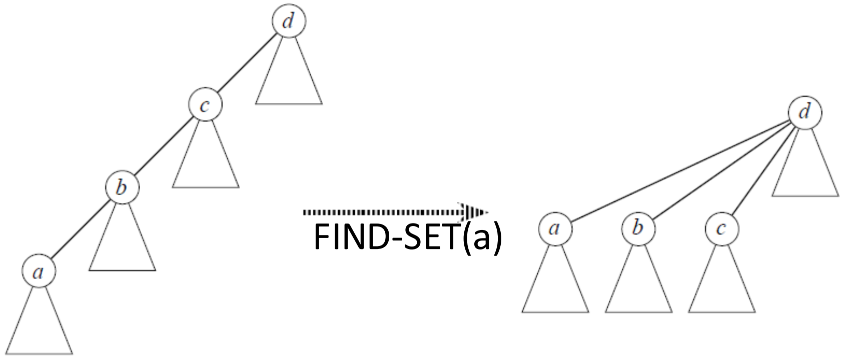
c.rank = 2, f.rank = 2, arbitrarily choose c or f as the parent and increase its rank (f.rank=3), depending on implementation.

Disjoint-Set Forests

- **FIND-SET**: follow pointers to the root. Every node along the search path to x will be directly connected to the root by the end of the operation, called **path compression**.

```
FIND-SET($x$)
  if $x \neq x.p$
      $x.p$ = FIND-SET($x.p$)
  return $x.p$
```

// Make a pass up to find the root, and a pass down as recursion unwinds to update **each node on the find path to point directly to root**.



Path compression can make the tree much flatter than the rank suggests, but the rank is intentionally **not decremented.**

Disjoint-Set Forests

Running Time (Sec 19.4 proof out of scope)

The running time of all disjoint-set forest operations starting from an empty graph with $|V|$ vertices to a graph with $|E|$ edges is $O\big(E\alpha(V)\big)$, where $\alpha(V)$ is a very slowly growing function.

In all practical situations, $\alpha(V) \leq 4$. Thus, we can view the running time as almost linear in $|E|$.

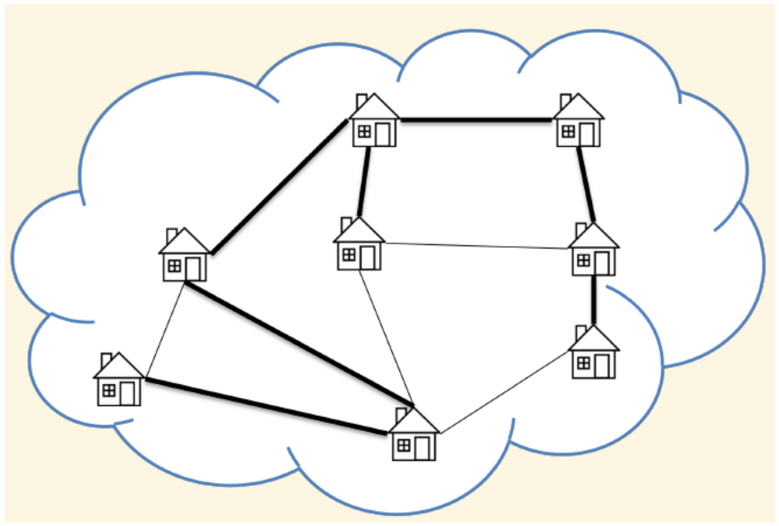
Minimum Spanning Tree

- Spanning Tree
- Kruskal’s Algorithm
- Prim’s Algorithm

Spanning Tree

- A town has a set of houses and a set of roads. A road connects 2 and only 2 houses.
- ***Goal:*** Repair enough (and no more) roads such that

  - everyone stays connected: can reach every house from all other houses.
- ***Definition:*** Spanning tree is a set of edges forming a tree and connecting all nodes in a graph.



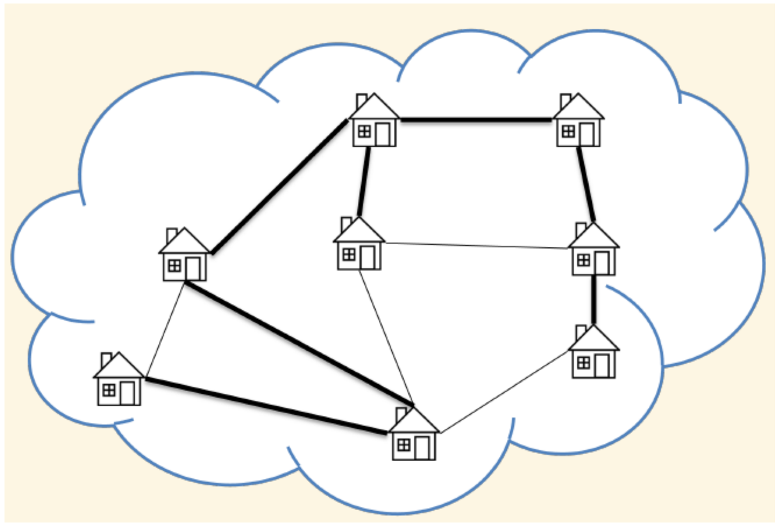
Tree: A special type of graph that is connected and has no cycles.

Minimum (Cost) Spanning Tree

- A town has a set of houses and a set of roads. A road connects 2 and only 2 houses. A road connecting houses u and v has a repair cost $w(u, v)$.
- ***Goal:*** Repair enough (and no more) roads such that

  1. everyone stays connected: can reach every house from all other houses, and

  2. total repair cost is **minimum**.



Minimum Spanning Tree

Model as a graph:

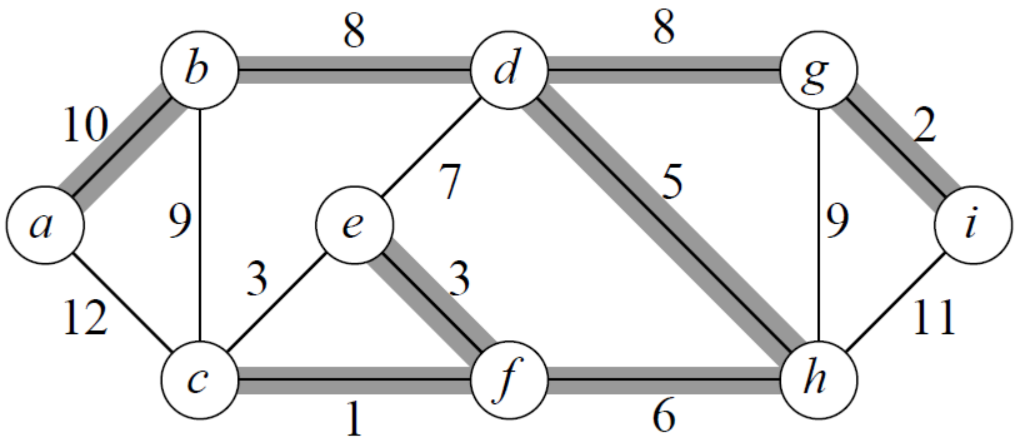
- Connected and undirected graph $G = (V, E)$.
- Weight $w(u, v) \rightarrow \mathbb{R}$ on each edge $(u, v) \in E$.
- Find subgraph as a tree $T \subseteq E$ such that
  1. T connects all vertices, and
  2. $w(T) = \sum_{(u,v)\in T} w(u, v)$ is minimized.

A spanning tree T whose weight is minimum over all spanning trees is called a minimum spanning tree (MST).

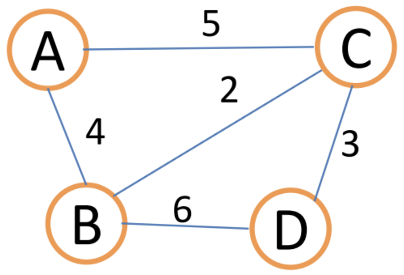
Minimum Spanning Tree

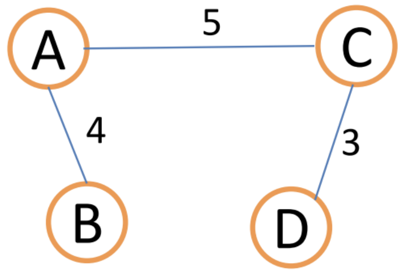
Some properties of an MST (marked in shade):

- It has $|V| - 1$ edges.
- It has no cycles, must be a tree.
- It might not be unique, e.g., replace edge (e, f) by (c, e) is also MST, with the same total weight.
- Weight can be negative and positive.

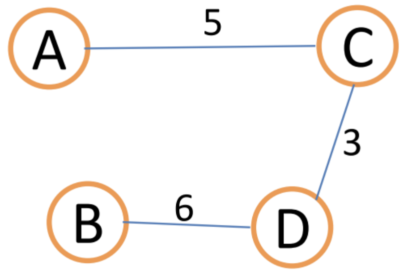


Example

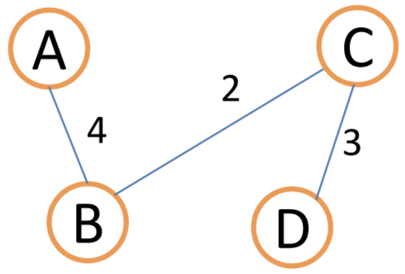




Cost: 12



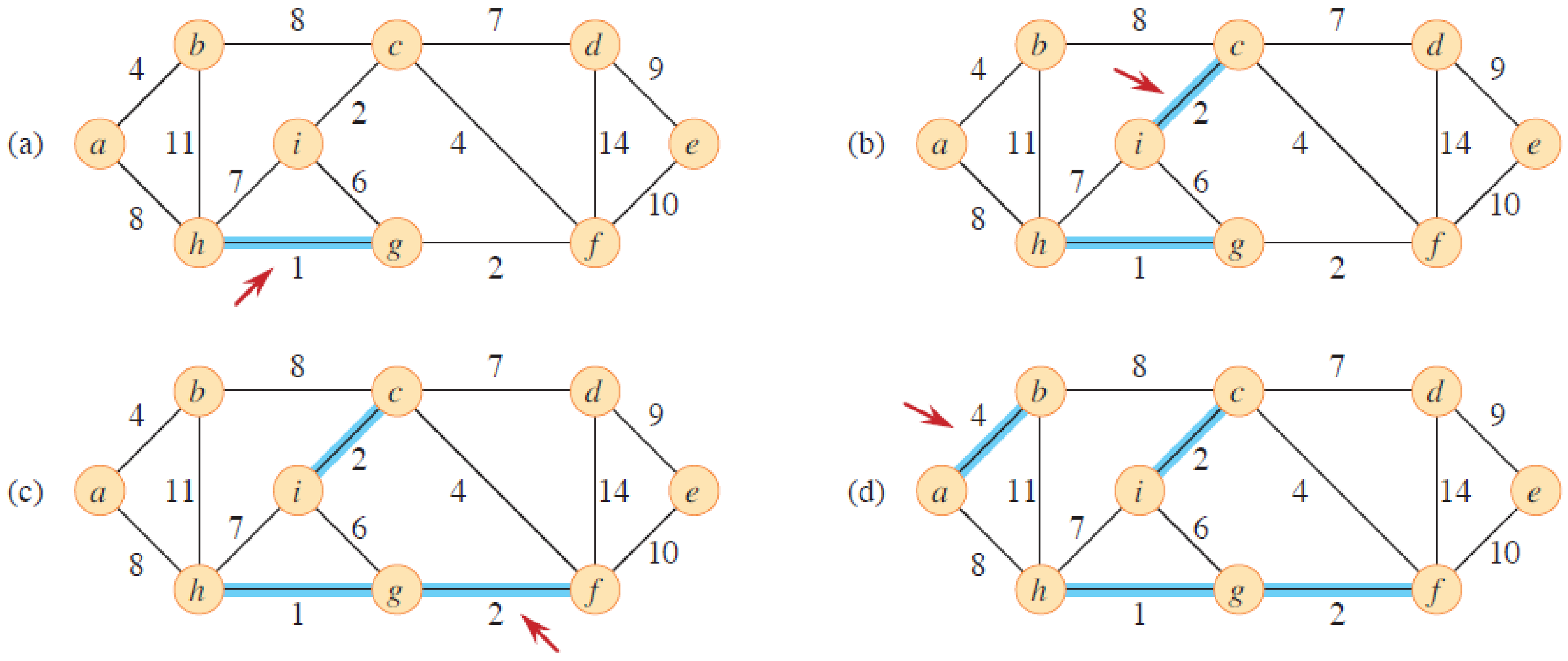
Cost: 14



Cost: 9

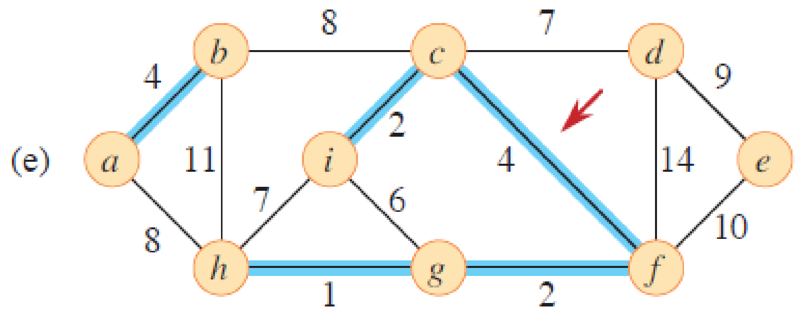
Kruskal’s Algorithm

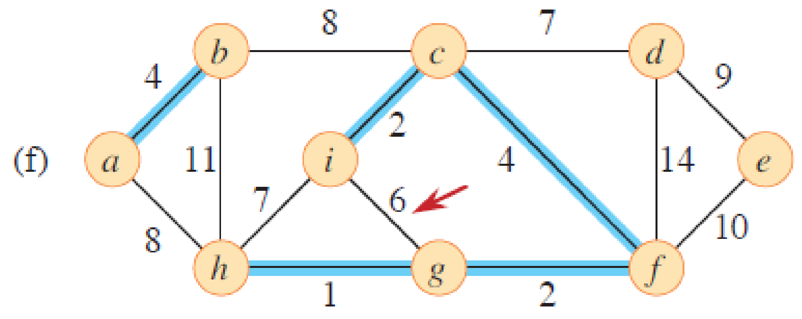
Illustration: Always select the smallest cost edge while not forming a cycle.



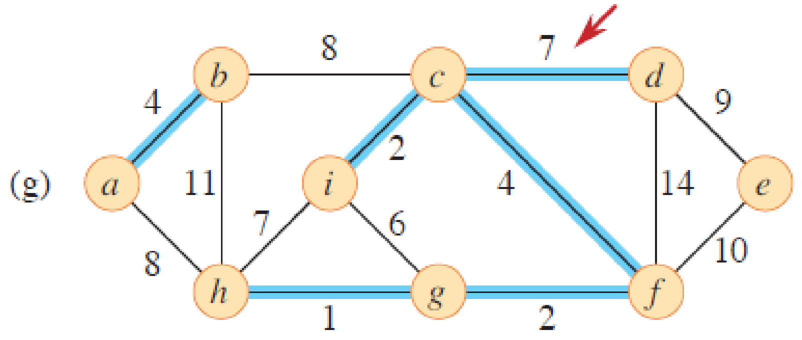
Kruskal's Algorithm

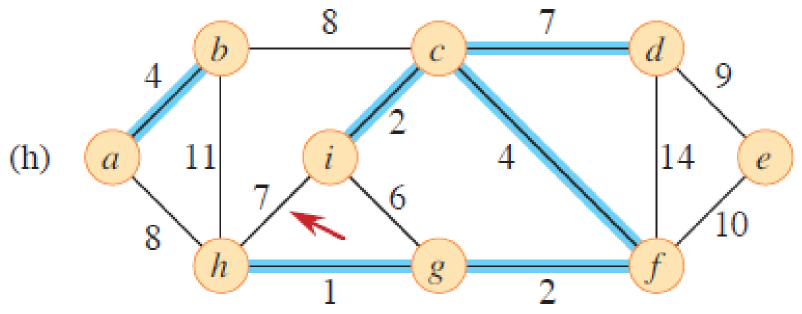
Illustration: Always select the smallest cost edge while not forming a cycle.





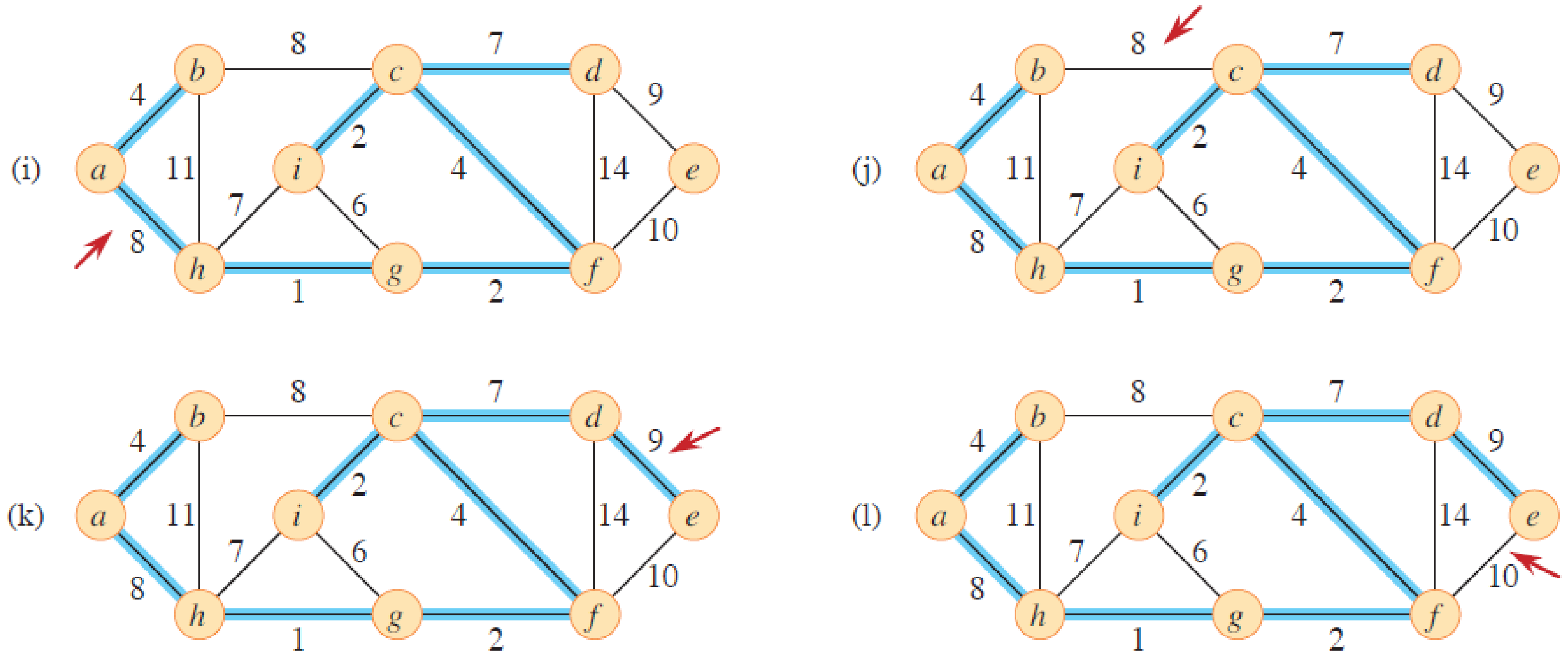
Don't select, otherwise form a cycle.





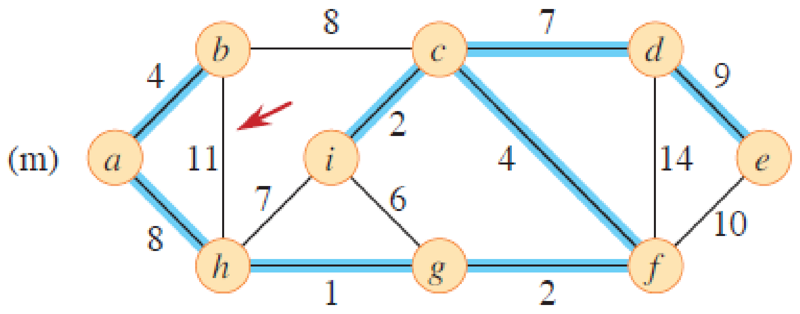
Kruskal’s Algorithm

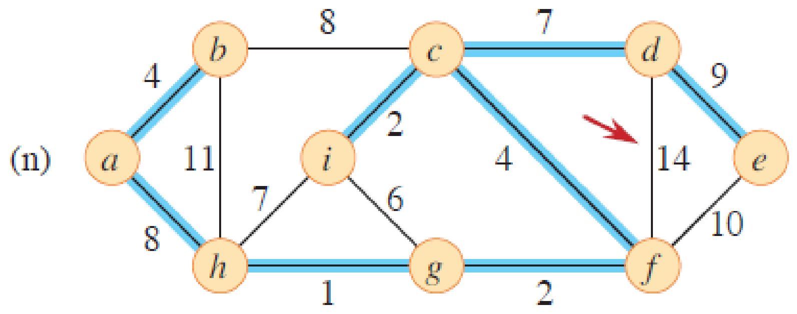
Illustration: Always select the smallest cost edge while not forming a cycle.



Kruskal’s Algorithm

Illustration: Always select the smallest cost edge while not forming a cycle.





Kruskal’s Algorithm

Pseudocode

```
KRUSKAL($G, w$)
  $A = \emptyset$        // Maintain a set $A$ of edges that forms a forest (i.e., a set of trees) at all times
  for each vertex $v \in G.V$
      MAKE-SET($v$)
  sort the edges of $G.E$ into nondecreasing order by weight $w$
  for each $(u, v)$ taken from the sorted list
      if FIND-SET($u$) $\neq$ FIND-SET($v$)  // Cycles are avoided: u and v are in different
                                            trees by checking the representatives
          $A = A \cup \{(u, v)\}$// Add edge (u, v)
          UNION($u, v$)
  return $A$
```

Kruskal’s Algorithm

Time Complexity Analysis

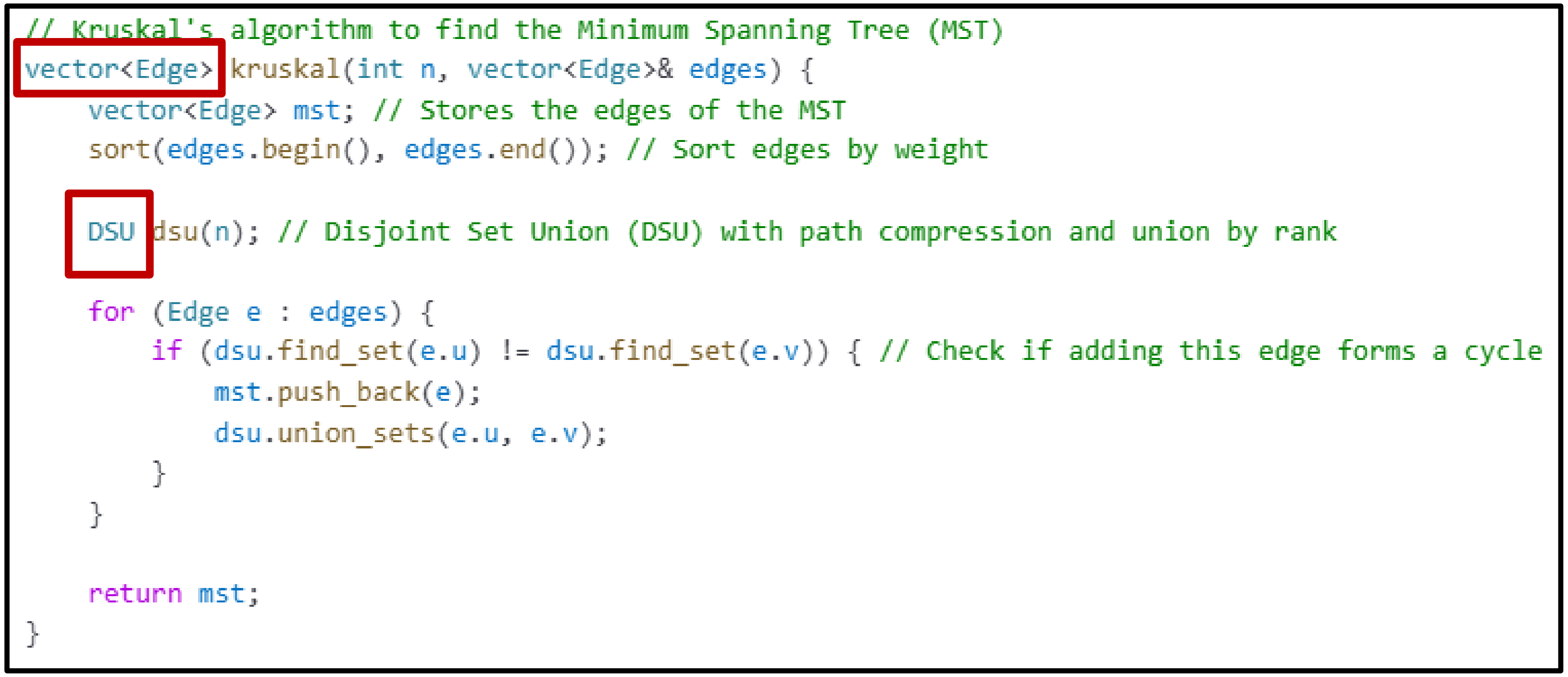
```
KRUSKAL($G, w$)
    $A = \emptyset$                                                       // Initialize $A$ $O(1)$
    for each vertex $v \in G.V$                                           // $O(V)$ MAKE-SETs
        MAKE-SET($v$)
    sort the edges of $G.E$ into nondecreasing order by weight $w$        // Sorting $O(E \log E)$
    for each $(u, v)$ taken from the sorted list
        if FIND-SET($u$) $\neq$ FIND-SET($v$)                             // FIND-SETs and UNIONs
            $A = A \cup \{(u, v)\}$
            UNION($u, v$)
    return $A$
```

Total time: $O(V) + O(ElogE) + O\big(E\alpha(V)\big) \Rightarrow \boldsymbol{O(E \log V)}$

- $|E| \ll |V|^2$ in simple graph so $O(logE) = O(logV)$, and $\alpha(|V|) = O(logV)$.
- |E|>|V|-1 in connected graph, O(V) becomes insignificant compared to $O(E \log V)$ when |E| is large.

Kruskal’s Algorithm

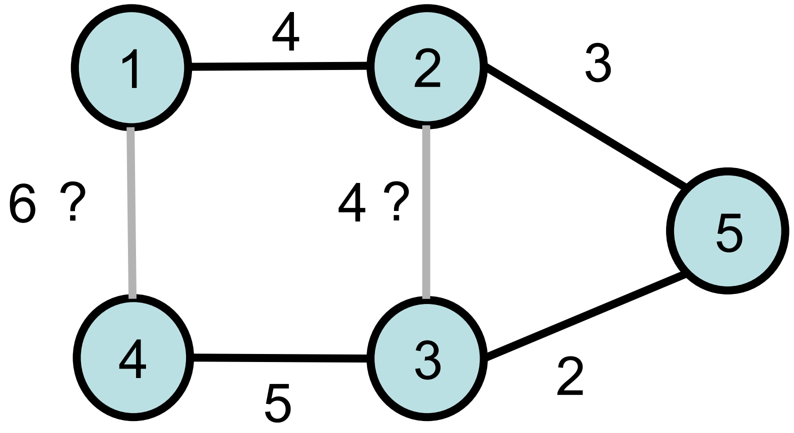
C++ Code



```cpp
// Kruskal's algorithm to find the Minimum Spanning Tree (MST)
vector<Edge> kruskal(int n, vector<Edge>& edges) {
    vector<Edge> mst; // Stores the edges of the MST
    sort(edges.begin(), edges.end()); // Sort edges by weight

    DSU dsu(n); // Disjoint Set Union (DSU) with path compression and union by rank

    for (Edge e : edges) {
        if (dsu.find_set(e.u) != dsu.find_set(e.v)) { // Check if adding this edge forms a cycle
            mst.push_back(e);
            dsu.union_sets(e.u, e.v);
        }
    }

    return mst;
}
```

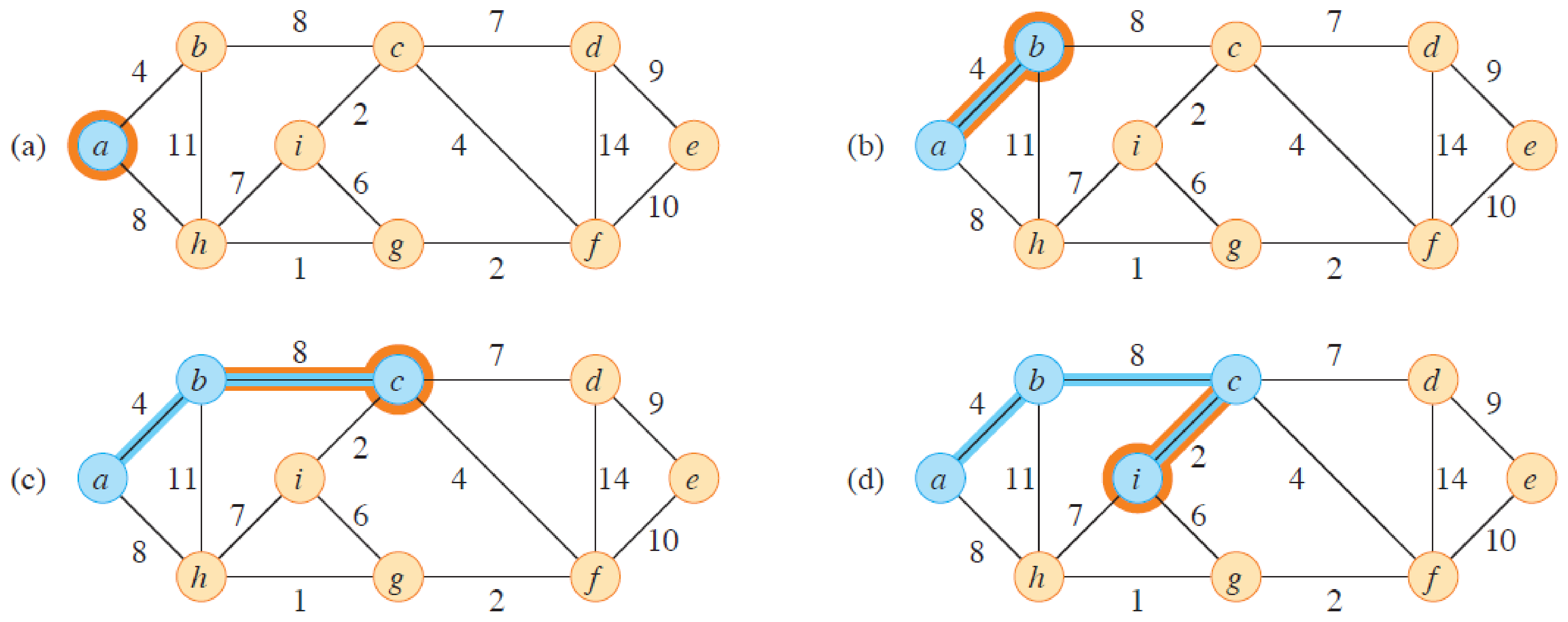
Exercise

Possible cost of the missing edges of the graph that are not in MST?



Prim’s Algorithm

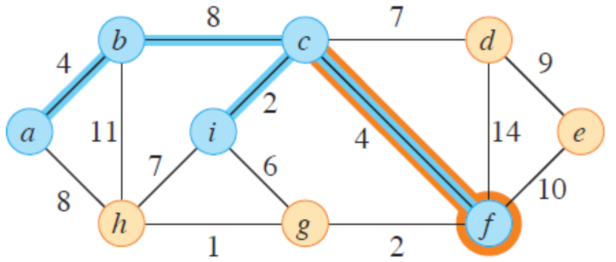
Illustration: Always select the smallest cost edge connected to already selected vertices – always a tree (no cycle).

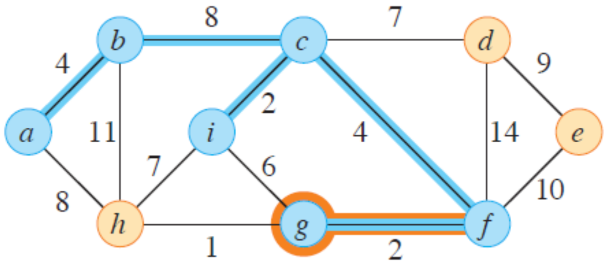


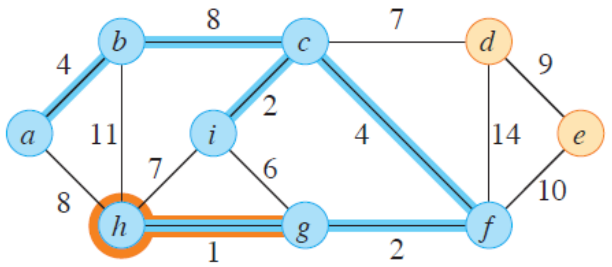
Prim’s Algorithm

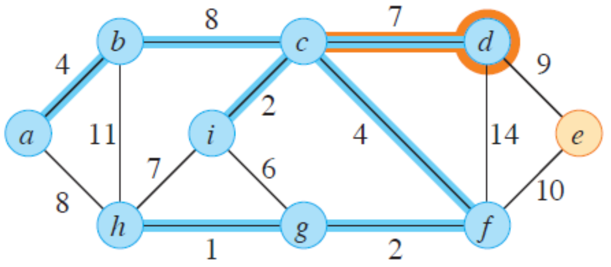
Illustration: Always select the smallest cost edge connected to already selected vertices – always a tree (no cycle).

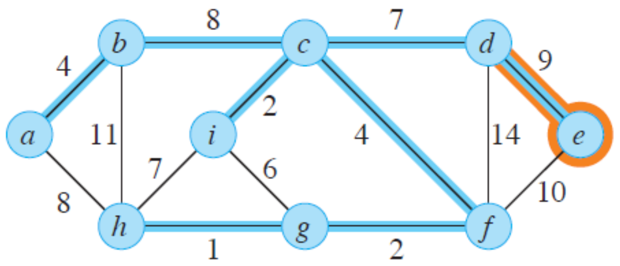
4 < 6

(e)


(f)


(g)


(h)


(i)


Prim's Algorithm

Pseudocode

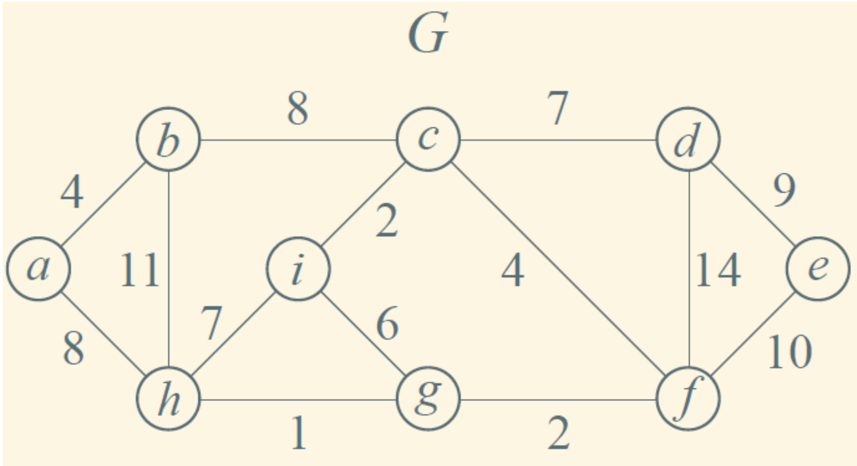
Tip 1: 'key' models the cheapest cost to connect v to the growing MST.

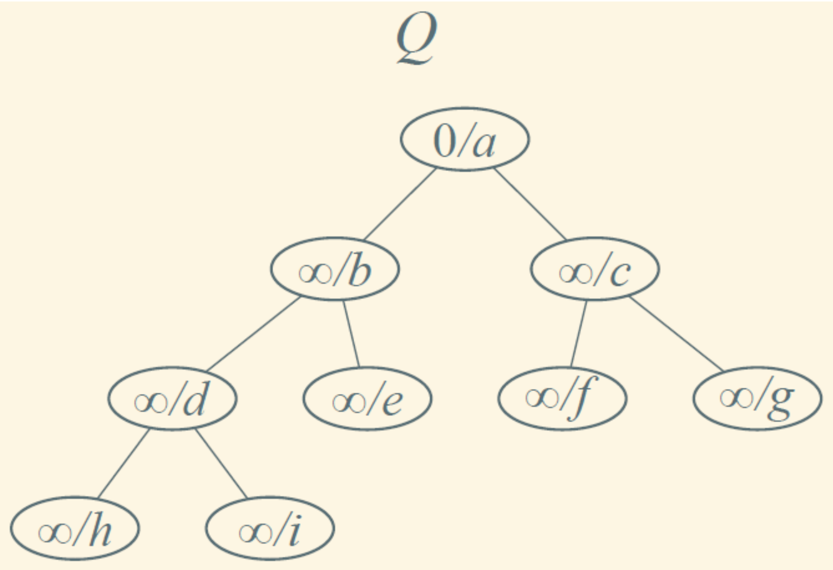
```
MST-PRIM($G, w, r$)

Initialization
1   for each vertex $u \in G.V$
2       $u.key = \infty$ // initialize minimum weight of any edge connecting $u$ to the tree
3       $u.\pi = \text{NIL}$ // parent of $u$
4   $r.key = 0$          // root
5   $Q = \emptyset$
6   for each vertex $u \in G.V$     // insert into min-heap (min-priority queue)
7       INSERT($Q, u$)

Selection by min-heap
8   while $Q \neq \emptyset$
9       $u$ = EXTRACT-MIN($Q$)          // add $u$ to the tree
10      for each vertex $v$ in $G.Adj[u]$   // update keys of $u$'s non-tree neighbors
11          if $v \in Q$ and $w(u, v) < v.key$   // compare after selected $u$
12              $v.\pi = u$
13              $v.key = w(u, v)$              // apply weight relaxation
14              DECREASE-KEY($Q, v, w(u, v)$)
```

Tip 2: Cycles are avoided inherently by removing the selected vertex from min-priority queue Q (line 9), thus it'll never be reselected from Q.

Prim’s Algorithm

Pseudocode – More Illustration

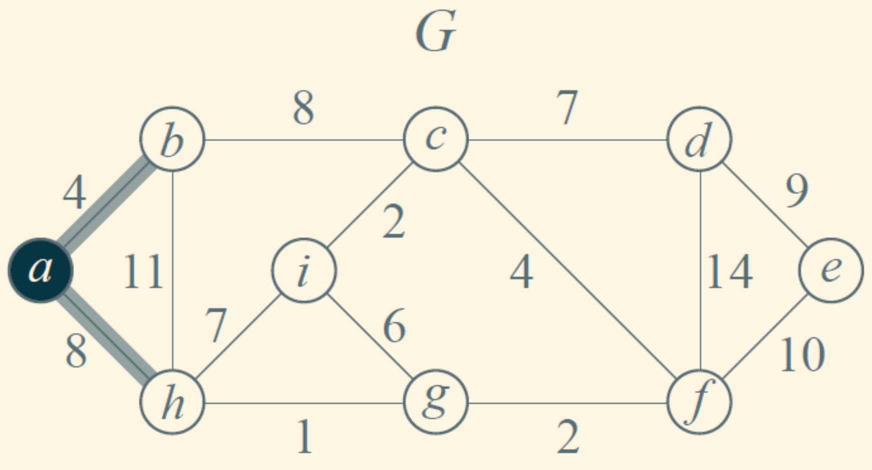


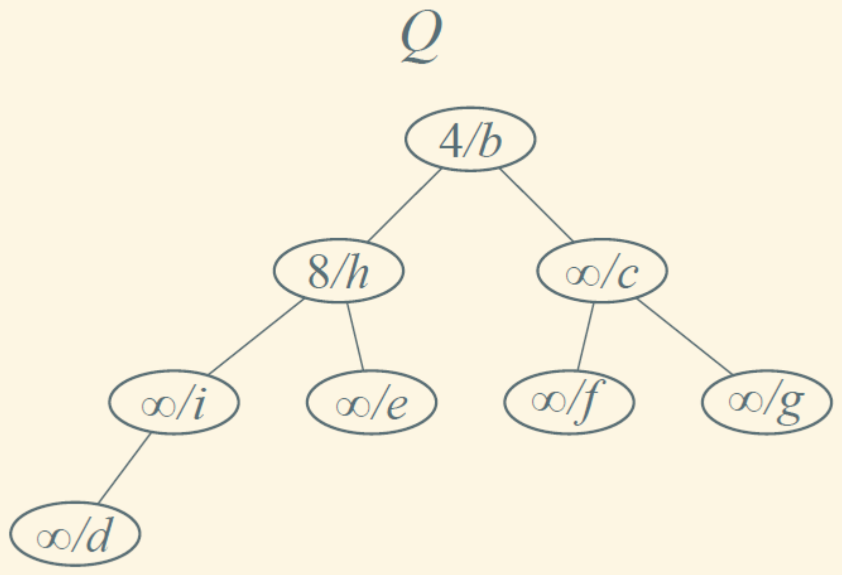


The number in each node represents the key. The letter indicates the vertex name.

Prim’s Algorithm

Pseudocode – More Illustration



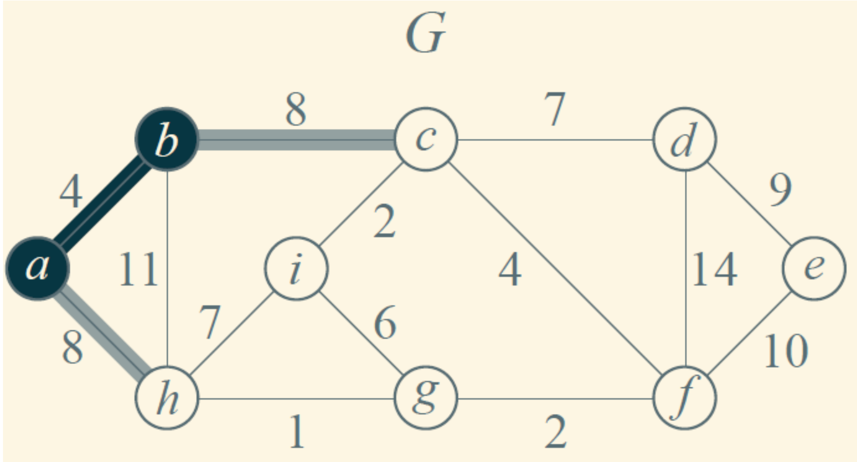


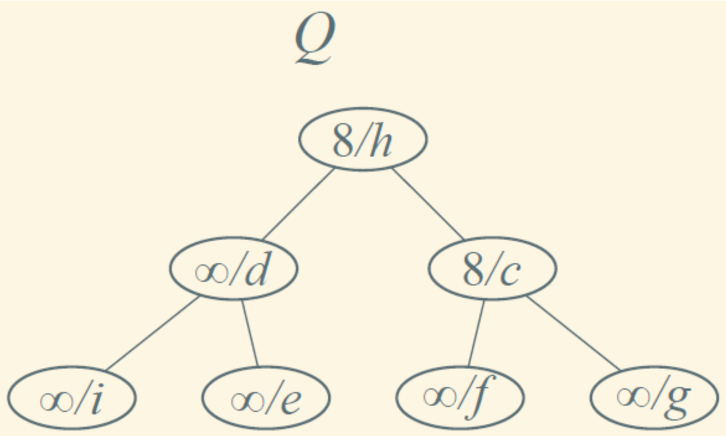
Highlighted edges indicate a parent-child relation.

The number in each node represents the key. The letter indicates the vertex name.

Prim's Algorithm

Pseudocode – More Illustration



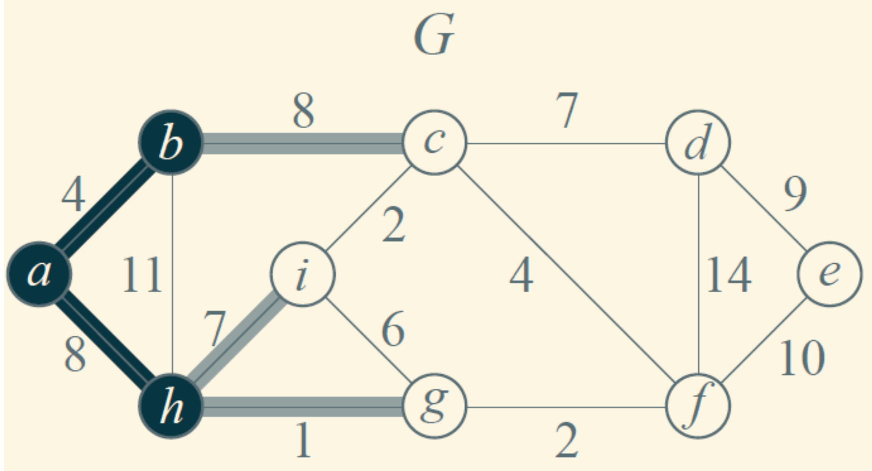


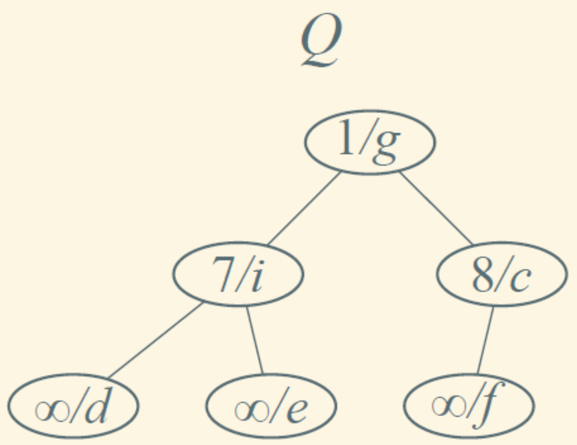
Highlighted edges indicate a parent-child relation. Black edges are part of a minimum spanning tree.

The number in each node represents the key. The letter indicates the vertex name.

Prim's Algorithm

Pseudocode – More Illustration



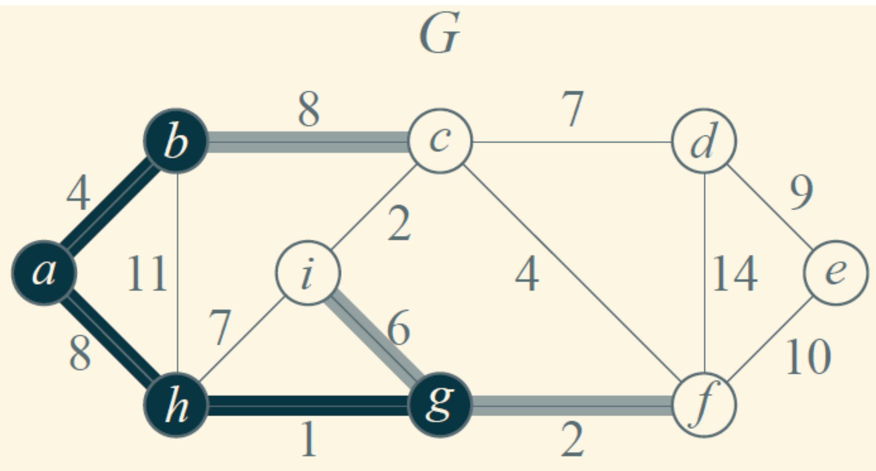


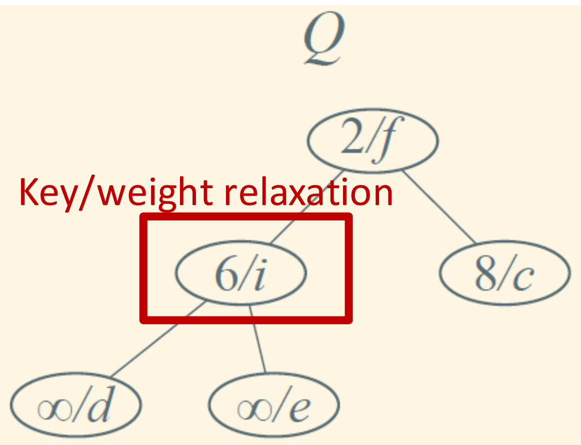
Highlighted edges indicate a parent-child relation. Black edges and nodes are part of a minimum spanning tree.

The number in each node represents the key. The letter indicates the vertex name.

Prim’s Algorithm

Pseudocode – More Illustration



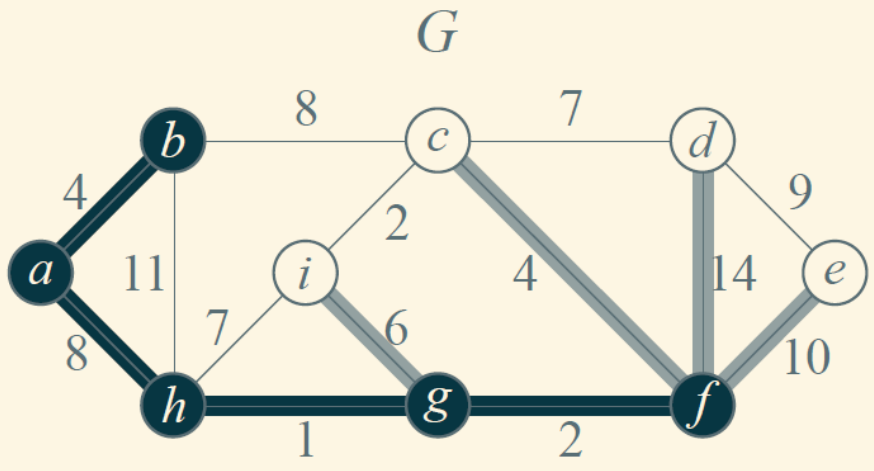


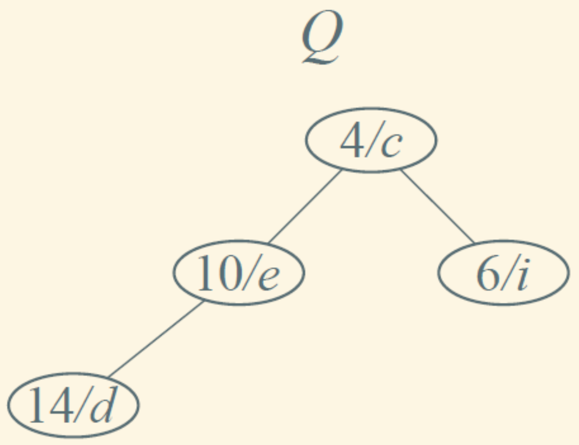
Highlighted edges indicate a parent-child relation. Black edges and nodes are part of a minimum spanning tree.

The number in each node represents the key. The letter indicates the vertex name.

Prim’s Algorithm

Pseudocode – More Illustration



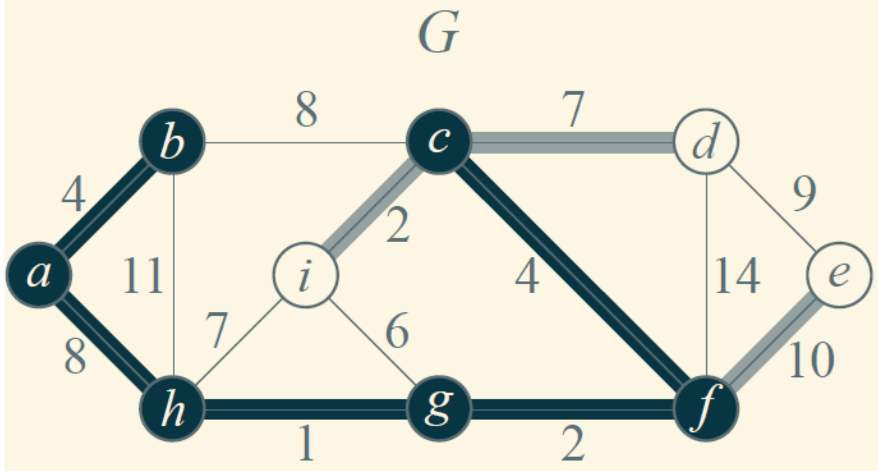


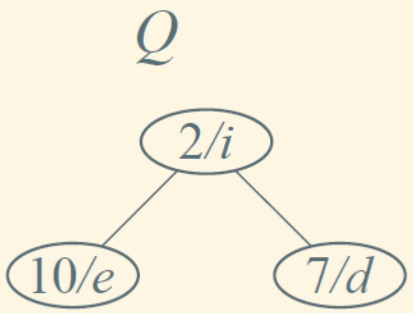
Highlighted edges indicate a parent-child relation. Black edges and nodes are part of a minimum spanning tree.

The number in each node represents the key. The letter indicates the vertex name.

Prim's Algorithm

Pseudocode – More Illustration



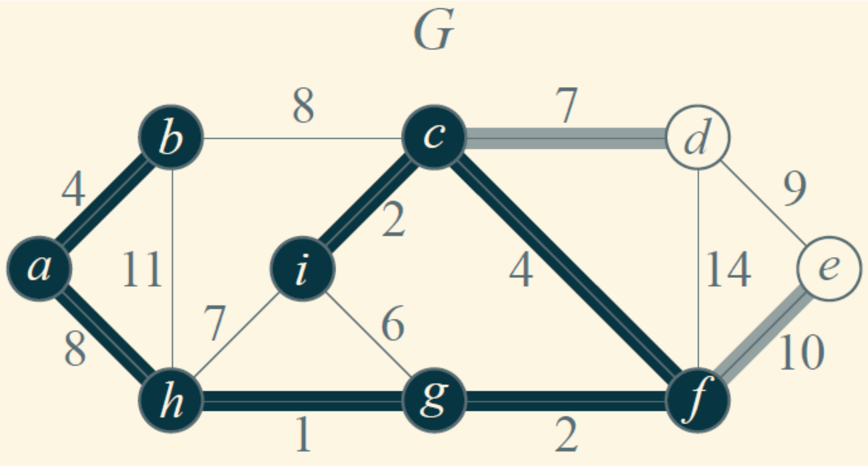


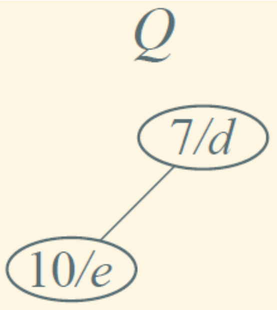
Highlighted edges indicate a parent-child relation. Black edges and nodes are part of a minimum spanning tree.

The number in each node represents the key. The letter indicates the vertex name.

Prim’s Algorithm

Pseudocode – More Illustration



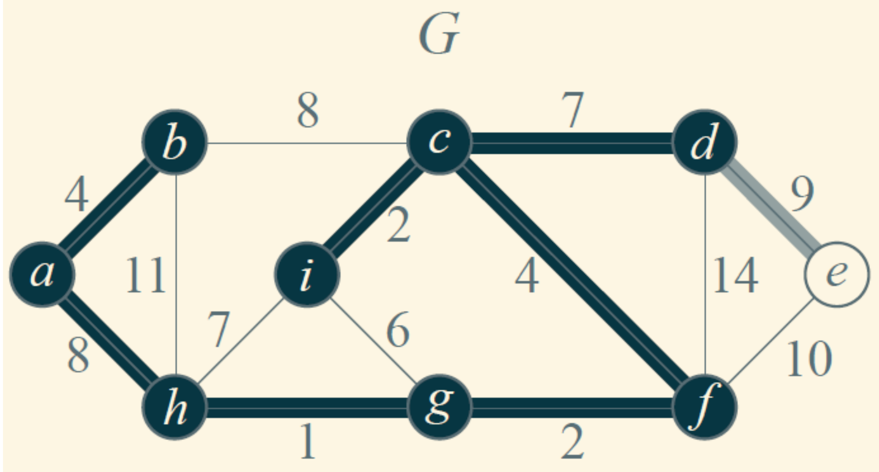


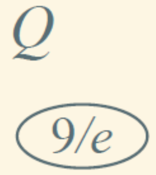
Highlighted edges indicate a parent-child relation. Black edges and nodes are part of a minimum spanning tree.

The number in each node represents the key. The letter indicates the vertex name.

Prim’s Algorithm

Pseudocode – More Illustration



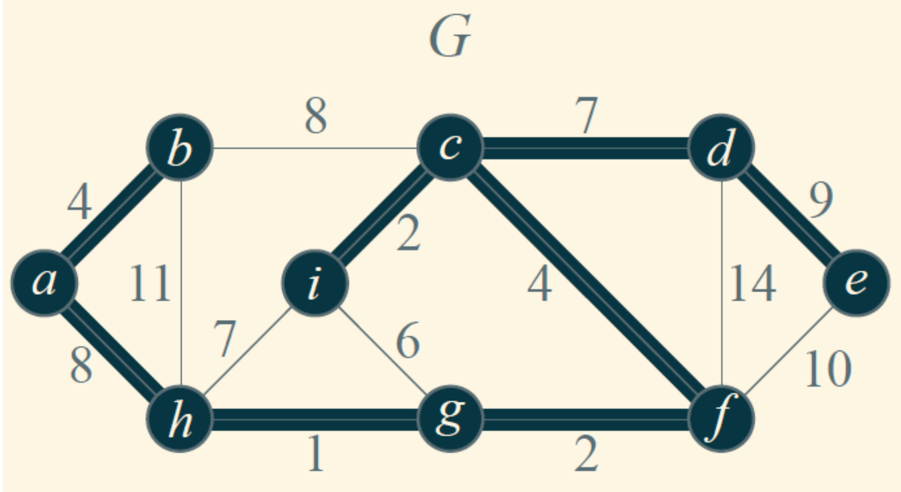


Highlighted edges indicate a parent-child relation. Black edges and nodes are part of a minimum spanning tree.

The number in each node represents the key. The letter indicates the vertex name.

Prim’s Algorithm

Pseudocode – More Illustration

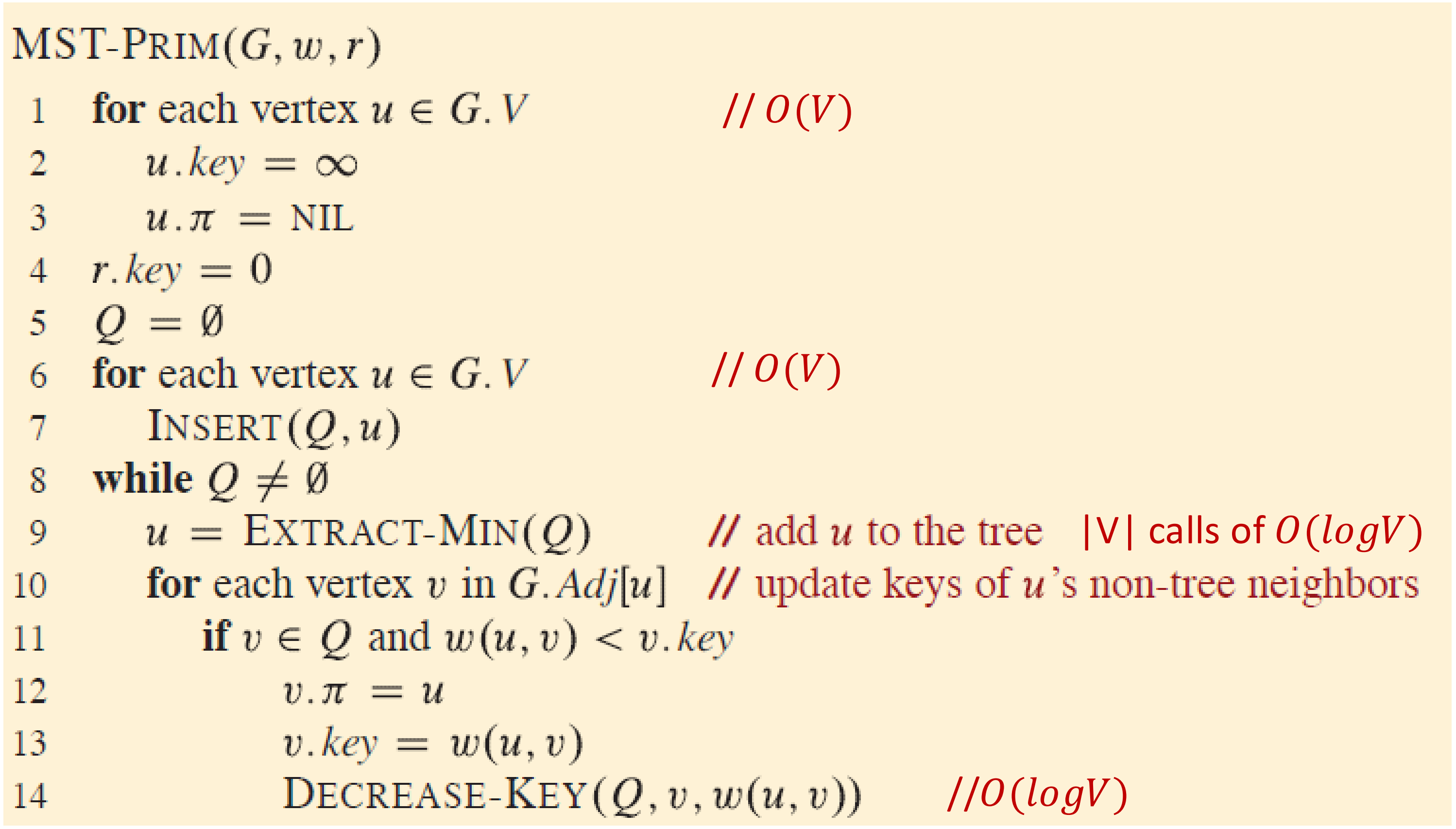


Q empty

Black edges and nodes are part of a minimum spanning tree.

Prim’s Algorithm

Time Complexity Analysis



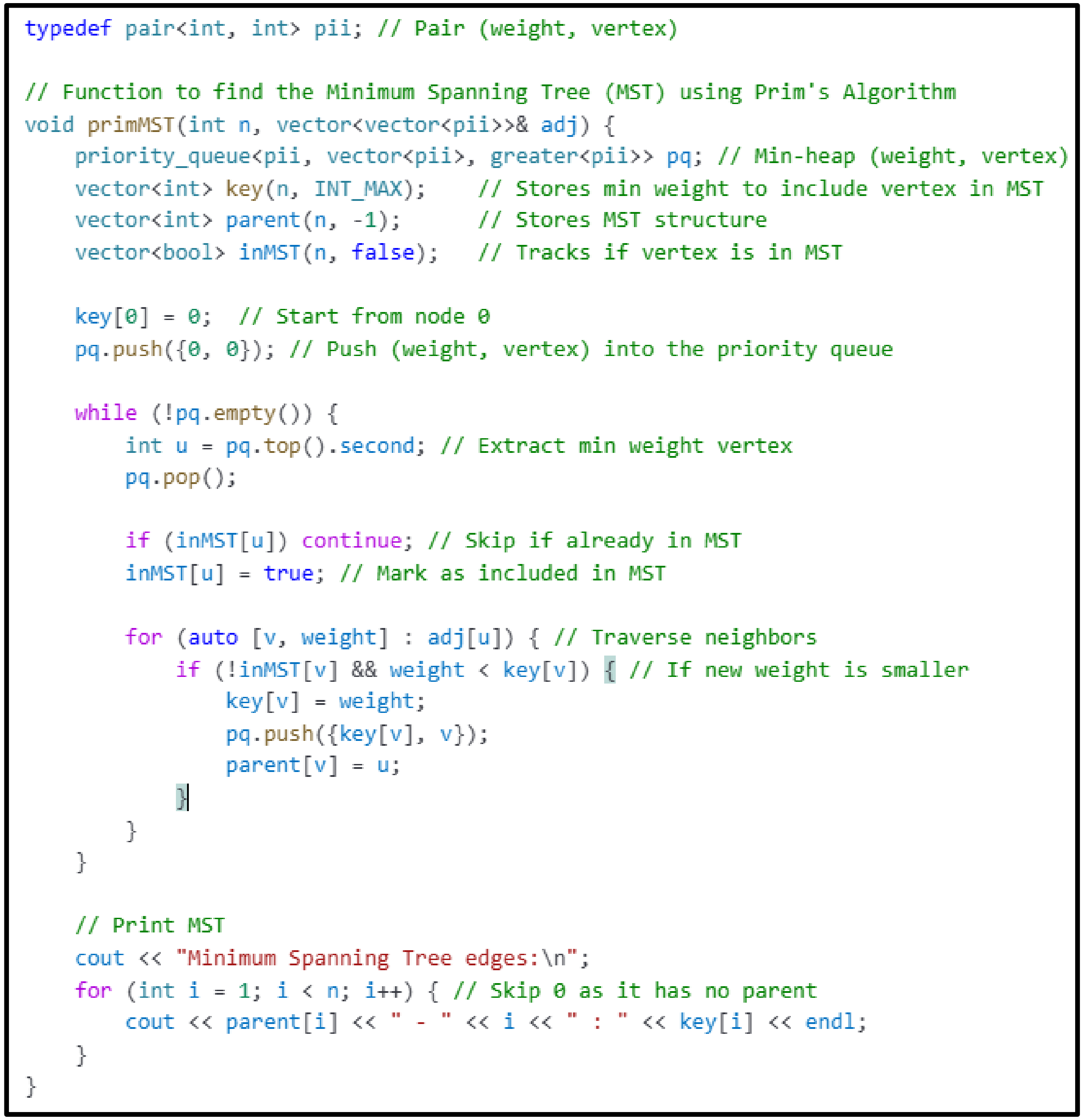
```
MST-PRIM($G, w, r$)
for each vertex $u \in G.V$                    // $O(V)$
    $u.key = \infty$
    $u.\pi$ = NIL
$r.key = 0$
$Q = \emptyset$
for each vertex $u \in G.V$                    // $O(V)$
    INSERT($Q, u$)
while $Q \neq \emptyset$
    $u$ = EXTRACT-MIN($Q$)            // add $u$ to the tree    |V| calls of $O(logV)$
    for each vertex $v$ in $G.Adj[u]$    // update keys of $u$’s non-tree neighbors
        if $v \in Q$ and $w(u, v) < v.key$
            $v.\pi = u$
            $v.key = w(u, v)$
            DECREASE-KEY($Q, v, w(u, v)$)        //$O(logV)$
```

$O(E)$ *for* loops, as $2|E|$ adjacent edges

Total time: $O\big((V + E)logV\big) \Rightarrow \boldsymbol{O(E\ logV)}$
because$|E| > |V| - 1$ (i.e., $|V| = O(E)$) for connected graph.

Prim’s Algorithm

C++ Code



```cpp
typedef pair<int, int> pii; // Pair (weight, vertex)

// Function to find the Minimum Spanning Tree (MST) using Prim's Algorithm
void primMST(int n, vector<vector<pii>>& adj) {
    priority_queue<pii, vector<pii>, greater<pii>> pq; // Min-heap (weight, vertex)
    vector<int> key(n, INT_MAX);    // Stores min weight to include vertex in MST
    vector<int> parent(n, -1);      // Stores MST structure
    vector<bool> inMST(n, false);   // Tracks if vertex is in MST

    key[0] = 0;  // Start from node 0
    pq.push({0, 0}); // Push (weight, vertex) into the priority queue

    while (!pq.empty()) {
        int u = pq.top().second; // Extract min weight vertex
        pq.pop();

        if (inMST[u]) continue; // Skip if already in MST
        inMST[u] = true; // Mark as included in MST

        for (auto [v, weight] : adj[u]) { // Traverse neighbors
            if (!inMST[v] && weight < key[v]) { // If new weight is smaller
                key[v] = weight;
                pq.push({key[v], v});
                parent[v] = u;
            }
        }
    }

    // Print MST
    cout << "Minimum Spanning Tree edges:\n";
    for (int i = 1; i < n; i++) { // Skip 0 as it has no parent
        cout << parent[i] << " - " << i << " : " << key[i] << endl;
    }
}
```

Summary

- Disjoint-set (union-find) data structure: efficiently merge for a dynamic graph.
- Minimum spanning tree (MST): connectivity and global optimization problem, connect all vertices with minimum cost from a connected, undirected, and weighted graph. The tree structure isn't unique but the total cost is the same.
- Kruskal's MST: always select the smallest cost edge while not forming a cycle, use disjoint-set, $O(E \log V)$ total time.
- Prim's MST: always select the smallest cost edge connected to the selected vertices, use min-heap (min-priority queue), $O(E \log V)$ worst-case time.